VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Absolvování individuální odborné praxe

Individual Professional Practice in the Company

2019 Jakub Holaza

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Jakub Holaza**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Absolvování individuální odborné praxe
Individual Professional Practice in the Company

Jazyk vypracování: čeština

Zásady pro vypracování:

1. Student vykoná individuální praxi ve firmě: Tieto Czech s.r.o.
2. Struktura závěrečné zprávy:
   a) Popis odborného zaměření firmy, u které student vykonal odbornou praxi a popis pracovního zařazení studenta.
   b) Seznam úkolů zadaných studentovi v průběhu odborné praxe s vyjádřením jejich časové náročnosti.
   c) Zvolený postup řešení zadaných úkolů.
   d) Teoretické a praktické znalosti a dovednosti získané v průběhu studia uplatněné studentem v průběhu odborné praxe.
   e) Znalosti či dovednosti scházející studentovi v průběhu odborné praxe.
   f) Dosažené výsledky v průběhu odborné praxe a její celkové zhodnocení.

Seznam doporučené odborné literatury:

Podle pokynů konzultanta, který vede odbornou praxi studenta.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **doc. RNDr. Petr Šaloun, Ph.D.**

Konzultant bakalářské práce: Mgr. Zdeněk Dřízga

Datum zadání: 01.09.2018

Datum odevzdání: 30.04.2019

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 4. dubna 2019 .....

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v bakalářských programech VŠB-TU Ostrava.

V Ostravě 4. dubna 2019

Tieto Czech s.r.o.
28. října 3346/91
702 00 Ostrava - Moravská Ostrava
IČO 64608051 DIČ CZ64608051

Týmto by som sa veľmi rád poďakoval spoločnosti Tieto Czech s.r.o. za poskytnutie možnosti absolvovania bakalárskej praxe na pozícii *software developer*. Rád by som poďakoval môjmu kolegovi z MM tímu Tomašovi Mariňákovi, kolegom zo stažistického tímu a Mgr. Zdeňkovi Dřízgovi, bez ktorých by táto práca nevznikla.
V neposlednom rade by som chcel poďakovať vedúcemu práce doc. RNDr. Petru Šalounovi, Ph.D. za vedenie a pomoc pri spracovaní tejto práce.

Abstrakt

V tejto bakalárskej práci popisujem odbornú prax v spoločnosti Tieto Czech s.r.o., priebeh a uplatnenie mojích vedomostí na vykonávaných projektoch.
V prvej časti práce popisujem históriu a predstavenie spoločnosti Tieto Czech s.r.o. V druhej časti sa venujem rozboru mojej histórie v Tieto Czech s.r.o., popis môjho pôsobenia vo firme a na projektoch. V tretej časti sa venujem rozboru vývoja projektu a popisu použitých technológií, táto časť je rozdelená na tri časti. V prvej sa venujem fáze plánovania vývoja projektu. V druhej časti sa venujem vývoji back-end a v tretej sa zameriavam na vývoj front-end. V druhej a tretej časti taktiež popisujem použité technológie v danej časti. V poslednej časti sa venujem uplatneným a chýbajúcim znalostiam počas odbornej praxe.

Kľúčové slová: odborná prax, vývoj, back-end, front-end, Tieto, Java, Spring Framework, React, Redux, Maven, Hibernate, JPA, koncový bod, TypeScript, Bootstrap

Abstract

In this bachelor thesis I describe my professional practise at the company Tieto Czech s.r.o., its course and applying of knowledge on projects.
In the first part I introduce Tieto Czech s.r.o. and describe its history. In the second part I describe my position at Tieto Czech s.r.o. and I also describe projects that I have been working on. In the third part I present development of projects, I describe used technologies, this part is divided into three parts. In the first sub-part I describe the phase of the planning of the development of the project. In the second sub-part I present back-end development and in the third part I describe font-end development. In the second and the third sub-part I also describe technologies used in these parts. In the very last part I am writing about the skills I used and the skills I lacked during my professional practise.

Key Words: professional practise, development, back-end, front-end, Tieto, Java, Spring Framework, React, Redux, Maven, Hibernate, end point, TypeScript, Bootstrap

Obsah

Zoznam použitých skratiek a symbolov

REST	–	Representational state transfer
API	–	Application programming interface
JSON	–	JavaScript object notation
SQL	–	Structured Query Language
HQL	–	Hibernate Query Language
HTML	–	Hypertext mark-up language
CSS	–	Cascading style sheets
RSS	–	Rich site summary
HTTP	–	Hypertext transfer protocol
UI	–	User interface
UX	–	User experience
JPQL	–	Java persistence query language
JPA	–	Java persistence api

Zoznam obrázkov

Zoznam tabuliek

Zoznam výpisov zdrojového kódu

1 Úvod

Pre vykonávanie odbornej praxe vo firme som sa rozhodol z viacerých dôvodov. Najdôležitejším však bolo to, že som chcel nadobudnúť nové skúsenosti z oblasti praktického vývoja, a tým doplniť teoretické a praktické vedomosti nadobudnuté zo školy a zo súkromných projektov.

Počas môjho pôsobenia v Tieto Czech s.r.o. som sa dozvedel o množstve technológií, metód vývoja a postupoch, o ktorých som dovtedy nevedel. Mal som možnosť pracovať v tímoch so skúsenými vývojármi, a tým sa od nich učiť nové praktiky. Taktiež som mal možnost komunikovať s ľudmi zo zahraničia, a tým zlepšiť svoje znalosti anglického jazyka.

V spoločnosti Tieto Czech s.r.o. som pracoval na pozicií *software developer*. Počas môjho pôsobenia v tejto firme som už mal možnost vývoju softvéru, ktorý popisujem ďalej v práci. V tejto práci sa venujem popisu aplikácie služiacej pre potreby personálneho oddelenia – práca s dátmi zamestnancov. Základné požiadavky sú evidencia dostupných zamestnancov podľa ich rolí, preferencií a kompetencií vhodných pre projekty, evidencia hľadaných pracovných pozicií pre projekty a nástroj pre párovanie vyššie spomínaných evidenicí a hľadanie ideálnej zhody medzi voľnými pozíciami a dostupnými zamestnancami.

V tejto práci popisujem projekt, na ktorom som počas svojej odbornej praxe pracoval, fungovanie v rámci nadnárodnej spoločnosti, analýzu požiadaviek na aplikáciu, špecifikácie funkčných a nefunkčných požiadaviek, návrh architektúry, dizajn aplikácie – návrh riešenia a jeho implementáciu.

2 O Tieto Czech s.r.o.

2.1 O Tieto

Tieto je najväčšia škandinávska IT spoločnosť, ktorá v Ostrave a v Brne zamestnáva viac ako 2600 ľudí. Podieľajú sa na projektoch pre severských zákazníkov, za ktorými pravidelne každý mesiac až 200 z nich cestuje. Tieto má vlastný startupový program Tieto Nerds, v rámci ktorého podporuje kreatívne nápady českých inovátorov a spojuje ich s biznisom svojich zákazníkov. Spoločnosť je známa svojou severskou kultúrou založenou na rešpekte, dôvere a vyváženosti pracovného a osobného života. Dlhodobo patri medzi päť vysnívaných IT spoločností českých študentov [1].

2.2 História Tieto

História Tieto korporácie siaha do roku 1968. Vtedy firma niesla názov Tietotehdas Oy. V roku 1999, kedy sa spojila fínska spoločnosť Tieto a švédska spoločnosť Enator, došlo k zmene mena na TietoEnator a v roku 2010 opäť na Tieto. Spoločnosť neukončila svoje pôsobenie v Švédsku, ale zmena súvisí s novou stratégiou spoločnosti a má signalizovať zjednodušenie [2].

Obr. 1: Tieto Logo

3 Moje pôsobenie v Tieto Czech s.r.o.

3.1 Pracovné zaradenie

Vo firme pracujem od marca roku 2018. Predtým, než som bol prijatý, som musel prejsť dvomi kolami prijímacieho riadenia. Prvé kolo bolo zložené z komunikácie s HR v českom a anglickom jazyku, to bolo prvé využitie anglického jazyka v praxi, potom z písomného testu zloženého z otázok zameraných prevažne na problematiku jazyka Java. Druhé kolo bolo zložené z pohovoru s manažérom Mgr. Zdeňkem Dřizgou, kde sme sa rozprávali o mojich doterajších projektoch a skúsenostiach. Po nastúpení do práce bolo potrebné pripraviť si počítač nainštalovaním a konfiguráciou programov potrebných pre prácu.

Vo firme pracujem na pozícií *software developer*. Moja práca je konkrétnejšie zameraná na vývoj prevažne *web-based* aplikácií. Tieto aplikácie sú zväčša zložené z 3 častí.

1. Dátový zdroj – databáza
2. Back-end
3. Front-end

Dátovy zdroj

Dátový zdroj, prevažne databáza je množina štruktúrovaných dát a slúži na uloženie informácií takým spôsobom, že počítačový program, alebo človek môže použiť špeciálny jazyk na získavanie týchto informácií. Vďaka presne určenej štruktúre umožňuje ľahké vyhľadávanie a triedenie aj pri veľkom množstve dát [4]. Ako dátový zdroj najčastejšie využívame relačné databázy, ale veľmi často ich kombinujeme s objektovými, poprípade grafovými databázami.

Back-end

Back-end je označenie pre skrytú časť webovej aplikácie (*užívateľ ju nevidí*). Väčšinou zodpovedá za komunikáciu s databázou, obsluhovanie požiadaviek a prácu s dátami. Na vývoj back-endu využívame programovací jazyk Java a k nemu populárny *open-source framework Spring Framework*.

Front-end

Front-end je označenie pre časť webovej aplikácie, ktorú vidí bežný návštevník webu. Vývoj *front-endu* je prevažne vďaka skriptovaciemu programovaciemu jazyku JavaScript a vďaka knižnici k tomuto jazyku ReactJS.

3.2 Predchádzajúce projekty

Stážisti pracujú na projektoch, z ktorých je väčšina využívaná na interné účely firmy, to znamená, že tieto aplikácie sú určené pre vnútorné využitie v rámci firmy, napríklad aplikácia na správu zamestnancov. Počas mojej praxe som pracoval na rôznych projektoch popísaných nižšie.

3.2.1 R-finder

Popis aplikácie nám bol predstavený našim manažérom tak, že je potrebné spraviť aplikáciu, v ktorej bude možné hľadať vzťahy medzi zamestnancami, projektami, kompetenciami alebo manažérmi. Dôvod vývoja tejto aplikácie bolo zjednodušenie práce personálnemu oddeleniu pri hľadaní správnych zamestnancov do projektu. Úlohou bolo vytvoriť aplikáciu, kde sa dalo vyhľadať a vyfiltrovať zamestnancov podľa potrebných filtrov, a tým zistiť, že niektorí zamestnanci už spolu pracovali, a tým bude jednoduchšie vytvorenie tímu pre nový projekt.

3.2.2 Business Trip export

Pre vyplnenie dokladu o pracovnej ceste je potrebné vyplniť zložitý Excel súbor údajmi ohľadom pracovnej ceste. Z dôvodu zjednodušenia tohto procesu sme mali za úlohu vytvoriť aplikáciu, ktorá pomôže užívateľom s vyplňovaním týchto údajov do tohto Excelu. Úlohou bolo vytvoriť webovú aplikáciu, v ktorej si užívateľ v niekoľkých jednoduchých krokoch zadá údaje o svojej pracovnej ceste a aplikácia mu na základe týchto údajov vytvorí požadovaný Excel súbor.

4 O projekte

4.1 Požiadavky

V rámci spoločnosti existuje portál s názvom *MyStaffing*, kde manažéri alebo personálne oddelenie môže vytvárať ponuky na projekty, hľadať ľudí podľa ich preferovaných kompetencií, rolí, lokácie, a podobne. Jediná možnosť takéhoto vyhľadávania je postupným prechádzaním všetkých požiadaviek (*ďalej ako requirements*) a voľných zamestnancov ponúkajúcich svoje služby (*ďalej ako advertisements*).

O celkovom projekte a o jeho požiadavkách sme sa dozvedeli od manažéra na zasadaní, s tým že na žiadosť od personálneho oddelenia je potrebné spraviť aplikáciu, vďaka ktorej bude možné jednoducho hľadať voľných ľudí na pracovné pozície v rámci spoločnosti. Hlavná funkčnosť celej aplikácie má byť to, že k pracovnej pozícii bude možné jedným klikom nájsť ideálnych kandidátov, a taktiež naopak, nájsť ku kandidátovi vhodné pracovné pozície.

4.2 Kľúčové otázky týkajúce sa informácií

4.2.1 Čo?

Informácie, s ktorými sme mali pracovať boli z počiatku nejasné. Jediné, čo sme vedeli, bolo to, že musíme nejakým spôsobom nájsť zhody medzi *requirements* a *advertisements*. Dôležitá otázka bola taktiež to, ako sa k informáciám dostaneme. Mali sme dve možnosti:

1. RSS feed – rozobrať a analyzovať RSS feed, dáta z neho postupne zmapovať na modely programovacieho jazyka a vložiť ich do databázy projektu.

2. Získať prístup k centrálnej databáze a odtiaľ dáta zduplikovať do databázy projektu.

Po niekoľkých konzultáciách a sedeniach s manažérom a kolegami, sme dospeli k názoru, že najlepšie bude využitie centrálnej databázy firmy. Najdôležitejším dôvodom bolo to, že v centrálnej databáze je viac užitočných dát ako v RSS feed-e.

4.2.2 Ako?

Keď už sme mali rozhodnuté odkiaľ budeme získavať informácie, potrebovali sme navrhnúť ako ich získať a ako s nimi pracovať. Vzhľadom na to, že aplikácia potrebuje aktuálne informácie k správnemu hľadaniu a filtrovaniu, tak sme potrebovali vyriešiť aktuálnosť dát. Došli sme k riešeniu, že sa budú dáta kopírovať z centrálnej databázy do databázy aplikácie, vždy po zapnutí aplikácie a potom v pravidelných intervaloch každých 30 minút.

4.2.3 Kde?

S aplikáciou sa pracuje na troch miestach:

- Personálne oddelenie pri hľadaní správnych zamestnancov na voľné pracovné pozície.
- Manažéri pri hľadaní voľných ľudí pre projekty.
- Zamestnanci s voľnou kapacitou, ktorí hľadajú pracovné miesto.

4.2.4 Kto?

Každý užívateľ s aplikáciou pracuje v roli užívateľa, pri čom môže prechádzať všetky *requirements* a *advertisements*. Každý užívateľ môže tieto dáta filtrovať, hľadať v nich a hľadať medzi nimi zhody.

4.2.5 Kedy?

S informáciami sa pracuje v prípadoch, keď treba nájsť správneho zamestnanca na danú voľnú pozíciu alebo pri hľadaní voľnej pozície pre zamestnancov, ktorí hľadajú projekty.

4.2.6 Prečo?

Dôležité je uľahčiť proces hľadania medzi stovkami záznamov bez možnosti filtrovania a prehľadávania. Taktiež je dôležité prispôsobiť informácie do stavu, kedy sú ľahko prehľadávateľné, filtrovateľné a je možné medzi nimi rýchlo a efektívne hľadať zhody.

5 MatchMaker

Matchmaker je výhradne interná aplikácia slúžiaca pre spoločnosť Tieto Czech s.r.o., ktorá slúži na zjednodušenie vyhľadávania vhodných *requirements* a *advertisements* pomocou filtrov a hľadaním zhôd medzi *requirements* a *advertisements*.

Funkčnosť aplikácie je rozdelená do troch častí.

5.1 Requirements

Stránka, na ktorej sú zobrazené v tabuľkovom formáte *requirements* dáta. Je tu možnosť medzi nimi vyhľadávať, filtrovať alebo si prečítať podrobnejšie informácie. Taktiež je tu možnosť k danému *requirement*-u nájsť najviac zhodné *advertisements*.

Táto stránka slúži najmä pre personálne oddelenie pri hľadaní dostupných projektov a pre zamestnancov, ktorí hľadajú projekty kam by sa mohli pridať.

5.2 Advertisements

Stránka, podobná stránke *requirements*, len s tým rozdielom, že na stránke sú zobrazené dáta ohľadom *advertisements*. Medzi dátami je možné hľadať, filtrovať ich, a tiež je možné k nim hľadať vyhovujúce *requirements*.

Táto stránka slúži pre personálne oddelenie alebo pre manažérov pri hľadaní správnych zamestnancov na projekt. Pomocou jednoduchých filtrov si môžu vytvoriť obraz ideálneho kandidáta.

5.3 Find match

Táto stránka kombinuje *requirements* stránku a *advertisements* stránku dohromady. Dáta sú v dvoch tabuľkách vedľa seba, čiže pri hľadaní zhody sú vidieť všetky dáta, čím sa urýchľuje možnosť vyhľadávania zhôd medzi zamestnancami a projektami.

Táto stránka slúži hlavne pre personálne oddelenie, pri potrebe rýchleho prehľadávania a hľadania zhôd medzi *requirements* a *advertisement*.

6 Riešené úlohy pri návrhu aplikácie

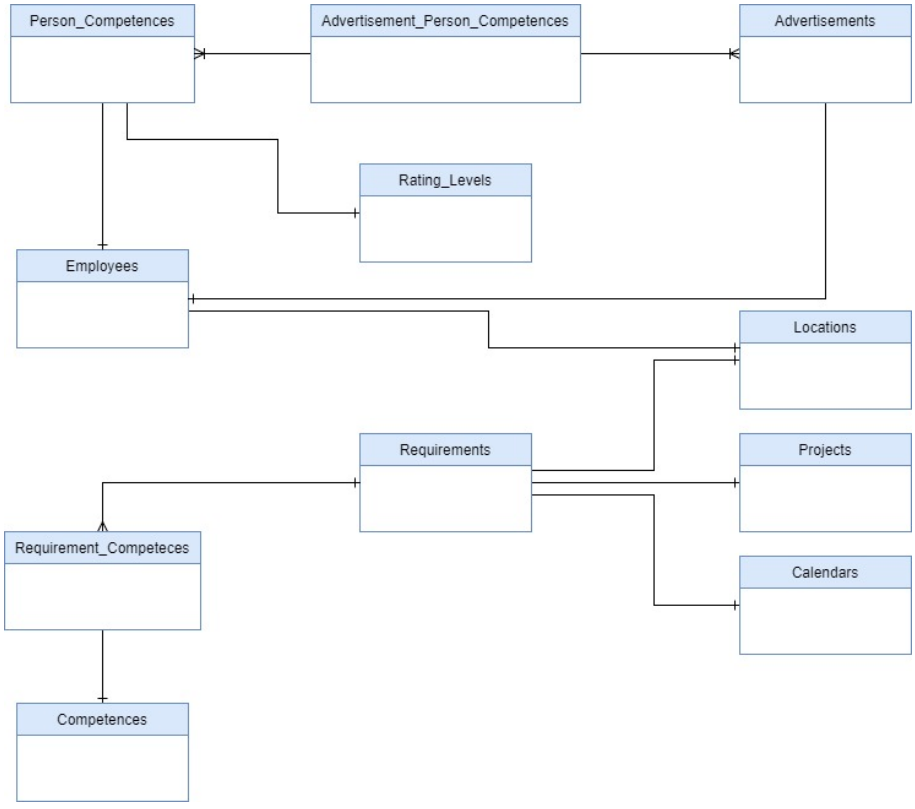
6.1 Návrh architektúry systému

Zadanie

Navrhnúť model aplikácie, spôsob fungovania a závislosti. Vybrať technológie, pomocou ktorých bude aplikácia vyvíjaná. Navrhnúť jednoduché užívateľské rozhranie, ktoré bude spĺňať všetky špecifikácie a požiadavky na aplikáciu.

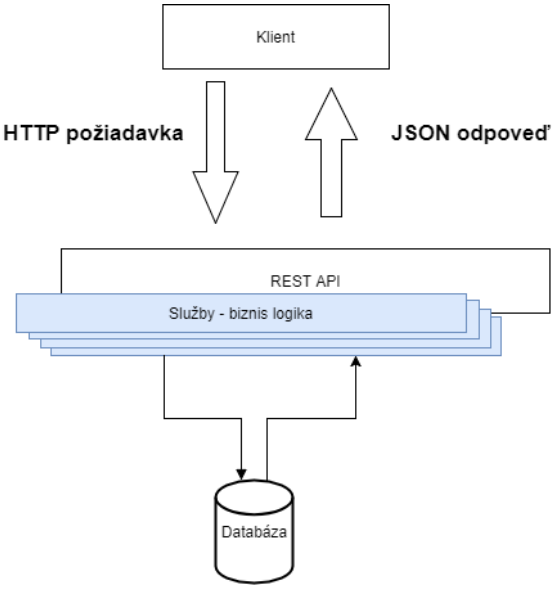
Riešenie problému

Návrh modelu aplikácie nám bol z veľkej časti uľahčený práve tým, že sme sa rozhodli kopírovať dáta z centrálnej databázy spoločnosti. Vďaka tomuto sme nemuseli riešiť dátový model a vzťahy medzi modelmi.



Obr. 2: Dátovy model (dátove polia vynecháne z dôvodu stručnosti)

Pre tento typ projektu sme sa rozhodli využiť *RESTful API* architektúru, pozri obr. 3. Ako dátové úložisko sme sa rozhodli použiť objektovo-relačný databázový systém PostgreSQL 11.2. Pre back-end sme sa rozhodli využiť technológie programovacieho jazyka Java a *framework* Spring Boot 2.0. Pre front-end sme sa rozhodli využívať JavaScript knižnice ReactJS a Redux.



Obr. 3: Zjednodušená schéma RESTful architektúry

6.2 Dizajn užívateľského rozhrania a UX dizajn

Zadanie

Návrh užívateľského rozhrania, tak aby splňovalo všetky špecifikácie a požiadavky na aplikáciu. Aby bol návrh jednoduchý pre uživateľov, tzv. *user friendly*.

Riešenie problému

V rámci niekoľkých zasadaní a konzultácií sme dospeli k tomu, že najlepšie bude aplikáciu vyhotoviť v tabuľkovej forme. Rozhodli sme sa že budeme smerovať k dizajnu, ktorý dovolí jednoduché hľadanie a filtrovanie v tabuľke.

6.3 Prístup do centrálne databázy spoločnosti

Zadanie

Získať prístup k potrebným dátam a informáciám potrebným k fungovaniu aplikácie.

Riešenie

Problém bol získať prístup dátam. Bolo potrebné vytvoriť niekoľko tiketov k tomu, aby sme získali tieto prístupy a na vyriešenie sa tiež čakalo niekoľko týždňov. Nakoniec sa nám podarilo získať pohľad (*view*) do potrebných tabuliek v centrálnej databáze.

7 Technológie využité pri implementácii back-end

Java

Java je objektovo orientovaný programovací jazyk. Je to taktiež výpočetná platforma, ktorá bola vydaná roku 1995 spoločnosťou Sun Microsystems. V súčasnosti je vyvíjaný spoločnosťou Oracle. Jeho syntax vychádza z jazykov C a C++. Zdrojové programy sa nekompilujú do strojového kódu, ale do medzistupňa, tzv. „byte-code“, ktorý nie je závislý od konkrétnej platformy. Tento byte-code neskôr vykonáva a spracováva interpreter, Java Virtual Machine [3].

Spring Framework

Spring Framework poskytuje obsiahly programový a konfiguračný model, pre moderný vývoj podnikových Java-based aplikácií – nasadenie na akýkoľvek druh platformy. Kľúčový element Spring-u je jeho infraštruktúrna podpora na aplikačnom leveli: Spring sa zameriava na “inštalatérstvo” podnikových aplikácií tak, že tímy sa môžu sústrediť na biznis logiku aplikácie, bez zbytočný väzieb na špecifické prostredie nasadenia [5].

Tento Framework sme sa rozhodli použiť z dôvodu toho, že už s ním máme skúsenosti.

Apache Maven

Apache maven je nástroj pre správu, riadenie a automatizáciu buildov aplikácií. Aj keď je možné použiť tento nástroj pro projekty písané v rôznych programovacích jazykoch, podporovaný je predovšetkým jazyk Java [6].

Primárne ciele Maven-u sú

- Zjednodušenie procesu build-ovania aplikácií.
- Poskytovanie jednotného spôsobu build-ovania .
- Poskytovanie informácií o projekte.
- Poskytovanie smernice pre „best-practice“ vývoj aplikácií.
- Dovolenie transparentného pridávania nových funkcií.

Hibernate ORM

Uľahčuje riešenie otázky zachovania dát objektov aj po ukončení behu aplikácie. Je jednou z implementácií Java Persistence API (JPA). Hibernate poskytuje spôsob, pomocou ktorého je možné zachovať stav objektov medzi dvomi spustenými aplikácia. Hovoríme o tom ,že udržuje dáta perzistentné. Dosahuje toho pomocou ORM, čo znamená, že mapuje objekty z jazyka Java

na objekty relačnej databáze. K tomu používa buď mapovacie súbory alebo anotácie. Keď sú objekty uložené v databáze, dá sa na nich opytovať pomocou jazyku HQL (Hibernate Query Language), ktorý je odvodený z jazyka SQL a je mu veľmi podobný [7].

Framework sme sa rozhodli použiť preto, že veľmi uľahčuje prácu s databázou samotnou a celkovo aj rieši problémy mapovania entít na modely jazyka Java.

8 Riešené úlohy pri implementácii back-end

8.1 Získavanie dát

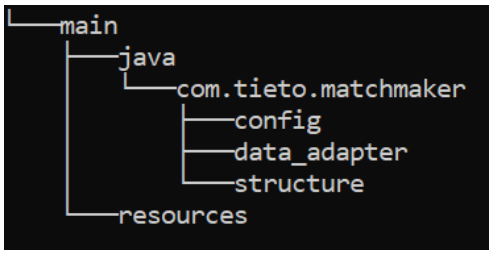
Zadanie

Zaistiť štruktúru projektu tak, aby bolo možné získať a skopírovať dáta z centrálnej databázy spoločnosti do súkromnej databázy projektu.

Riešenie

Prvý komplikovaný problém bol vyriešiť štruktúru projektu tak, aby bolo možné využívať počas jej funkcionality dva dátové zdroje – dve databázy. Vedeli sme, že centrálna databáza využíva databázový systém **Oracle** a databáza aplikácie je **PostgreSQL**. Pri práci so Spring Framework sa *properties* (v preklade vlastnosti), v tomto prípade údaje prístupu do databáz, z pravidla uvádzajú do súboru s príponou *.yml* alebo *.properties*. My sme zvolili možnosť YAML(.yml) súborov, teda *application.yml*. Našou úlohou bolo nastaviť konfiguráciu projektu tak, aby bolo možné jednu časť projektu pripojiť na **Oracle** – centrálnu databázu spoločnosti a druhú časť pripojiť na **PostgreSQL** – súkromnú databázu aplikácie. Rozhodli sme sa preto pre takúto štruktúru projektu, pozri obr. 4.

- data_adapter - časť pripojená na centrálnu databázu spoločnosti;
- structure - časť pripojená na databázu aplikácie;
- config - konfigurácia toho, aby boli tieto dve časti pripojené na správnu databázu;



Obr. 4: Štruktúra projektu

Konfigurácia dátovych zdrojov

Na konfiguráciu toho, aby structure a data_adapter boli pripojené na rôzne databázy bolo potrebné vytvoriť konfiguračné triedy **DataAdapterDbConfig** a **StructureDbConfig**. V týchto triedach je riešené načítavanie properties (v preklade *vlastnosti*) z *application.yml* súboru. Toto je robené spôsobom, že je do triedy pridaný objekt triedy **Environment** a pomocou anotácie *@Autowired* je zabezpečená *dependency injection*, čím je objekt z-inicializovaný a môžeme s ním

pracovať bez použitia *new*. Vďaka tomuto je načítanie properties veľmi jednoduché, pozri obr. 5 a 6. Kde *env* je objekt triedy **Environment** spomenutej vyššie.

```
dataSource.setUrl(env.getProperty("spring.datasource.url"));
dataSource.setUsername(env.getProperty("spring.datasource.username"));
dataSource.setPassword(env.getProperty("spring.datasource.password"));
```

Obr. 5: Nastavenie prístupu do databázy

```
# Oracle DB
spring:
  datasource:
    url: 'central.database.url'
    username: 'central.database.username'
    password: 'central.database.password'
    driver-class-name: oracle.jdbc.OracleDriver


# PostgreSQL DB
postgres:
  datasource:
    url: 'application.database.url'
    username: 'application.database.username'
    password: 'application.database.password'
    driver-class-name: org.postgresql.Driver
```

Obr. 6: application.yml - údaje prístupu k databázam

V týchto triedach som teda vytvoril *Java Bean* pomocou Spring Framework anotácie *@Bean*, čím zabezpečíme pre tieto Bean-y *dependency injection*.
Následne sú triedy **StructureDbConfig** a **DataAdapterDbConfig** o-anotované týmito anotáciami

```
@Configuration
@EnableTransactionManagement
@EnableJpaRepositories(basePackages = {"com.tieto.matchmaker.data_adapter"})
```

Výpis 1: Anotácie tried pre konfiguráciu dátoveho zdroja

Vďaka anotácií *@Configuration* Spring indikuje, že daná trieda obsahuje *@Bean* metódy a zabezpečí vygenerovanie *Spring Beans* a umožní ich použitie v aplikácii. Pre podporu transakcií v pripojení do databázy je používaná anotácia *@EnableTransactionManagment*. A pre túto konfiguráciu najdôležitejšie, anotácia *@EnableJpaRepositories* nastavená na požadovaný balíček (*package*), nám zabezpečí, že v danom balíčku budú používané Beans z tejto konfigurácie. Týmto sme zabezpečili to, že máme rozdelenú aplikáciu na dva dátové zdroje. Trieda **DataAdapterDbConfig** zabezpečuje pripojenie pre *data_adapter* časť projektu, teda pre pri-

pojenie k centrálnej databáze a trieda **StructureDbConfig** zabezpečuje pripojenie pre *structure* časť projektu.

8.2 Modely data_adapter časti

Zadanie

Je potrebné vytvoriť triedy a pridať k nim anotácie z frameworku JPA, ktoré budú použité ako DTO (Data Transfer Object) pre prenos dát z centrálnej databázy. Je teda nutné vytvoriť triedy tak, aby bolo možné namapovať dáta z databázy do objektov týchto tried. Pri správnom vytvorení triedy a pridaním anotácií frameworku JPA sa o všetok prenos a mapovanie stará konkrétna implementácia JPA framework, v našom prípade Hibernate ORM.

Poznámka autora

K všetkým tabuľkám bol možný prístup len pomocou pohľadu *(view)* do tabuľky, z dôvodu lepšej čitateľnosti budú väčšinou uvádzané názvy tabuliek bez predpony *XXHH_* a prípony *_V*.

Riešenie problému

Ako prvé bolo potrebné v *data_adapter* balíčku vytvoriť balíček, ktorý bude obsahovať pamäťové reprezentácie dát z databázy – modely. Vytvoril som teda balíček s názvom *models*.

Potom som začal vytvárať triedu RequirementDTO, ktorá mala slúžiť k prenosu základných dát o Requirement z centrálnej databázy spoločnosti z tabuľky(v tomto prípade šlo o pohľad do tabuľky) FEED_REQUIREMENTS. Aby JPA Framework využil túto triedu pre spojenie s pohľadom do tabuľky FEED_REQUIREMENTS bolo potrebné pridať nad definíciou triedy anotáciu *@Entity*, ktorá určila, že táto trieda bude JPA entitou. Ďalej bolo potrebné pridať nad definíciou triedy anotáciu *@Table* s pridaným atribútom *name*, ktorému bola nastavená hodnota "XXHH_FEED_REQUIREMENTS_V". To znamená, že tabuľka (v našom prípade pohľad do tabuľky) s názvom FEED_REQUIREMENTS bude v aplikácii reprezentovaná práve touto triedou. V prípade, že by sme neuviedli anotácii *@Table* žiadnu hodnotu atribútu *name*, ako základná hodnota by bol použitý názov triedy.

Ďalej bolo potrebné pridať triede premenné, ktoré budú obsahovať dáta z tabuľky. Najprv som začal s tým, že som vytvoril premennú typu Long s názvom *id* a pridal som jej anotáciu *@Id*. Vďaka tomuto JPA framework pozná, že sa jedná o primárny kľúč. Ďalej som tejto premennej pridal anotáciu *@Column*, kde som špecifikoval atribút *name* podľa názvu stĺpcu v tabuľke, v tomto prípade to bolo *"ID"*. V prípade, že nie je špecifikovaná hodnota tohto parametru, tak sa použije ako základná hodnota názov premennej.

Následne bolo potrebné pridať ďalšie premenné a k nim pridať anotácie, pri čom všetko bolo robené rovnakým spôsobom – vytvorenie premennej, anotácia *@Column* s atribútom *name*.

Trieda RequirementDTO má v databáze vzťahy aj s tabuľkami PA_COMPETENCES, LOCATIONS a PROJECTS. Tie v modeli RequirementDTO zatiaľ nevytvárame, lebo ešte nie sú vytvorené ich modely v aplikácií.

Podobne bolo potrebné vytvoriť triedy aj pre ostatné tabuľky z databázy. Rovnaký postup bol aplikovaný aj na tieto triedy

- AdvertisementDTO + @Table(name = „XXHH_ACTIVE_ADVERTISEMENTS_V“)
- CalendarDTO + @Table(name = „XXHH_CALENDARS_V“)
- EmployeeDTO + @Table(name = „XXHH_EMPLOYEES_MV“)
- LocationDTO + @Table(name = „XXHH_LOCATIONS_V“)
- PersonCompetenceDTO + @Table(name = „XXHH_PERSON_COMPETENCES_V“)
- RatingLevelDTO + @Table(name = „XXHH_RATING_LEVELS_V“)
- RequirementCompetenceDTO + @Table(name = „XXHH_PA_COMPETENCES_V“)

Ako je vidieť tabuľka Projects implementovaná nie je, pretože sme zistili, že táto tabuľka nám neposkytuje žiadne relevantné dáta, čiže ju vôbec v aplikácií nepotrebujeme.

Keď už boli implementované všetky modely, bolo možné pridať väzby medzi nimi. Vrátil som sa teda k triede RequirementDTO a pridal som tam premennú typu LocationDTO. Aby JPA Framework vedel, že sa jedná o väzbu medzi tabuľkami FEED_REQUIREMENTS a LOCATIONS, bolo potrebné pridať nad túto premennú anotáciu *@ManyToOne*, čím signalizujeme JPA Frameworku, že sa medzi týmito objektmi jedná o väzbu RequirementDTO n:1 LocationDTO, to znamená, že RequirementDTO má len jednu asociáciu na objekt triedy LocationDTO, ale jeden objekt triedy LocationDTO môže mať viac asociácií na objekty typu RequirementDTO. Táto anotácia obsahuje taktiež množstvo parametrov. Nás v tomto prípade zaujímal len jeden a to je atribút *fetch*. Tento atribút udáva to, či sa pri načítavaní objektu z databázy majú načítať aj dáta o tomto objekte. V našom prípade to znamená, či sa majú dáta o LocationDTO načítať pri načítavaní dát o RequirementDTO. Atribút *fetch* môže mať dve hodnoty, jednou je *FetchType.EAGER* (horlivo) a druhou *FetchType.LAZY* (lenivo). Pri *FetchType.EAGER* špecifikujeme frameworku JPA, že chceme tieto dáta načítavať zároveň s objektom RequirementDTO. Použitím *FetchType.LAZY* špecifikujeme, že dáta o LocationDTO načítavať nechceme, čiže pri načítavaní dát o RequirementDTO sa tieto dáta nenačítajú. Vzhľadom na to, že úlohou je skopírovať všetky dáta z centrálnej databázy, potrebovali sme, aby sa načítavalo všetko, preto sme pre tento prípad zvolili *FetchType.EAGER*. Anotácia *@ManyToOne* má nastavený atribút *fetch* na *FetchType.EAGER* štandardne.

Ďalej bolo potrebné pridať asociáciu na tabuľku PA_COMPETENCES, to znamenalo, že bolo potrebné pridať premennú typu RequirementCompetenceDTO, ale v tomto prípade to bolo iné, pretože RequirementDTO môže mať viac RequirementCompetenceDTO. Bolo potrebné teda

pridať List<RequirementCompetenceDTO> a k nemu pridať anotáciu *@OneToMany*. Táto anotácia bola použitá pre signalizovanie JPA Frameworku, že RequirementDTO môže mať viac RequirementCompetenceDTO, ale RequirementCompetenceDTO môže mať väzbu len na jeden RequirementDTO. Podobne, ako v prípade s LocationDTO a anotáciou *@ManyToOne*, sme chceli aby sa dáta načítavali spoločne, teda EAGER (horlivo). Anotácia *@OneToMany* však štandardne využíva *FetchType.LAZY* (lenivo), v tomto prípade sme museli atribút prepísať v anotácií *@OneToMany*(fetch = FetchType.EAGER). V tomto konkrétnom prípade trieda RequirementCompetenceDTO neobsahuje asociáciu na triedu RequirementDTO, ale obsahuje iba premennú typu Long s názvom projectAssigmentId, ktorá reprezentuje ID objektu triedy RequirementDTO. Preto bolo potrebné pridať anotácií *@OneToMany* aj atribút *mappedBy* s hodnotou *"projectAssigmentId"*, čím signalizujeme JPA Frameworku, že RequirementDTO je v triede RequirementCompetenceDTO reprezentovaný pod touto premennou.

K anotáciám *@OneToMany* a *@ManyToOne* existujú ešte anotácie *@OneToOne* a *@ManyToMany*. Pri anotácií *@ManyToMany* signalizujeme, že sa jedná o väzbu n : n. Toto je použité napríklad v triede AdvertisementDTO. Anotáciou *@OneToOne* signalizujeme, že sa jedná o väzbu 1 : 1, ale túto anotáciu v aplikácií nevyužívame.

8.3 Objekty pre prístup k dátam

Zadanie

Pri práci s databázou je nutné s databázou komunikovať, zvyčajne pomocou SQL (Structured Query Language) alebo pomocou iného vyhľadávacieho jazyka. V tomto prípade je to SQL. Mojou úlohou bolo vytvoriť DAO (*Data Access Object*), teda objekty prístupu k dátam.

Riešenie problému

S použitím Spring Framework a JPA Framework v kombinácií s Hibernate ORM je to však veľmi jednoduché a vo väčšine prípadov nie je vôbec nutné písať vyhľadávací jazyk. Budem popisovať vytváranie DAO objektov pre triedu RequirementDTO, rovnakým spôsobom sú vytvorené DAO objekty aj pre ostatné modely. Vytvoril som teda balíček s názvom *repositories* a v ňom rozhranie (*interface*), ktoré som nazval RequirementDTORepository. Z pravidla sú tieto rozhrania nazývané názvom modelu plus prípona *Repository*. Ďalej som toto rozhranie rozšíril (kľúčové slovo **extends**) o rozhranie JpaRepository, ktorému je nutné špecifikovať dátové typy. Keď sa bližšie pozrieme na rozhranie JpaRepository, tak zistíme že je to generické rozhranie s dvomi generickými dátovými typmi – JpaRepository<T, ID>. Prvý dátový typ je určený práve pre objekt, ktorým je tabuľka mapovaná do aplikácie, teda objekt s anotáciou *@Entity*, v tomto prípade RequirementDTO. Druhý dátový typ je určený pre špecifikovanie akého dátového typu je primárny kľúč tohto objektu. V tomto prípade je to Long. To však ešte nie je všetko. K tomu aby rozhranie správne fungovalo, je potrebné priamo nad definíciu pridať anotáciu *@Repository* zo Spring Framework. *Anotácia @Repository indikuje to, že rozhranie, ktoré obsahuje túto ano-*

*táciu je **Repository**, originálne definovaná v knihe Domain-Driven Design (Evans, 2003) ako „mechanizmus pre zabalenie ukladania, obnovovania a vyhľadávania, ktorý napodobňuje kolekciu objektov“ [8].* Vďaka anotácií *@Repository* Spring Framework deteguje, že sa jedná o Bean triedu a registruje ju do **ApplicationContext**. *ApplicationContext je centrálne rozhranie zabezpečujúce konfiguráciu pre aplikáciu. Toto rozhranie je počas behu aplikácie určené len na čítanie (read-only), ale môže byť znovu načítane, pokiaľ to implementácia aplikácie podporuje* [9].

```
@Repository
public interface RequirementDTORepository
   extends JpaRepository<RequirementDTO, Long>{
}
```

Výpis 2: RequirementDTORepository rozhranie ako DAO objekt

Keď sa bližšie pozrieme na rozhranie *JpaRepository<T, ID>*, tak zistíme, že obsahuje deklarácie metód pre vyhľadávanie, ukladanie, upravovanie a mazanie. Nás v tomto prípade zaujímajú len metódy pre vyhľadávanie. Ďalším dôležitým dôvodom, prečo rozširujeme rozhranie JpaRepository je ten, že toto rozhranie obsahuje anotáciu *@NoRepositoryBean*, vďaka ktorej dokáže Spring Framework počas behu aplikácie vytvoriť implementáciu k metódam tohto rozhrania a rozhraní, ktoré ju rozširujú. To znamená, že programátor vôbec nemusí písať implementáciu tohto rozhrania, ale implementácia sa vytvorí počas behu aplikácie.

So súčasnou implementáciou je možné pristupovať k objektom pomocou tohto rozhrania tým, že si dané rozhrania pomocou *dependency injection* načítame do aplikácie pri štarte a zavolaním metódy objektu tohto rozhrania máme prístup k dátam. Toto sa z pravidla vykonáva v *services* (v preklade *službách*), ktoré upravujú dáta v aplikácii ešte pred ich uložením do dátového úložiska ak je to potrebné, alebo upravujú tieto dáta po načítaní z dátového zdroja. Vytvoril som teda balíček *services* a v ňom rozhranie RequirementDTOService. Ako už je vidieť z názvu, využíva sa podobná konvencia ako pri tvorení *repository*, len s tým rozdielom, že sa používa prípona *Service*. V tomto som deklaroval metódu *findAll()* s návratovým typom *List<RequirementDTO>*. Ako názov napovedá, metóda bude vracať všetky záznamy RequirementDTO.

```
public interface RequirementDTOService {
   List<RequirementDTO> findAll();
}
```

Výpis 3: RequirementDTOService rozhranie

Dôvod toho, že sa vytvára rozhranie namiesto triedy je ten, že v budúcnosti je jednoduchšia zámena implementácie. Práve sa bude používať pripojenie k centrálnej databáze spoločnosti, ale v prípade, že by sa toto v budúcnosti menilo, tak je proces výmeny jednoduchší. Keď máme vytvorené rozhranie, vytvoríme v balíčku *services* balíček *impl*, ktorý bude obsahovať implementácie daných služieb (z balíčku *services*) rozhraní. V ňom som vytvoril triedu RequirementDTOServiceImpl, ktorá implementuje (kľúčové slovíčko **implements**) rozhranie

RequirementDTOService. Tejto triede som pridal anotáciu *@Service*, ktorá funguje podobne ako anotácia *@Repository*, a to v tom zmysle, že vďaka nej Spring deteguje, že sa jedna o Bean triedu a pri štarte aplikácie ju pridá do ApplicationContext. Taktiež vďaka tejto anotácii môžeme použiť anotáciu *@Autowired* nad premenné, set metódy a nad konštruktor. Vďaka *@Autowired* anotácii Spring deteguje, že objekt je potrebné nastaviť pomocou *dependency injection*. V tejto triede som vytvoril premennú typu RequirementDTORepository, ku ktorej som vytvoril tzv. *setter*. Setter je metóda, ktorou sa nastavuje premenná. Nad tento setter som pridal anotáciu *@Autowired*. Následne som prepísal metódy z rozhrania RequirementDTOService pomocou anotácie *@Autowired*, tak aby využívali metódy objektu rozhrania RequirementDTORepository.

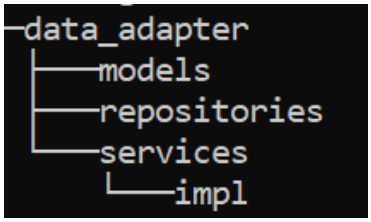
```
@Service
public class RequirementDTOServiceImpl implements RequirementDTOService {
   private RequirementDTORepository requirementDTORepository;

   @Autowired
   public void setRequirementDTORepository(RequirementDTORepository
       requirementDTORepository) {
      this.requirementDTORepository = requirementDTORepository;
   }

   @Override
   public List<RequirementDTO> findAll() {
      return requirementDTORepository.findAll();
   }
}
```

Výpis 4: Implementácia RequirementDTOService rozhrania

Týmto máme hotový prístup k objektom centrálnej databázy spoločnosti. Výsledná štruktúra *data_adapter* balíčku je na obr. 7.



Obr. 7: Štruktúra balíčku data_adapter

8.4 Modely a prístupové objekty k dátam structure časti

Zadanie

Podobne ako pre *data_adapter* časť, je potrebné vytvoriť modely a DAO (*objekty prístupu k dátam*) aj pre *structure* časť projektu. Z dôvodu toho, aby sme mohli skopírované dáta vložiť do databázy aplikácie a neskôr s nimi pracovať v rámci aplikácie.

Riešenie problému

Vzhľadom na to, že vytváranie modelov je takmer rovnaké ako pre *data_adapter* časť, popíšem len rozdiely. Triedy sme pomenovali rovnako ako *dáta_adapter* triedy, len bez prípony DTO. Kvôli tomu, aby sme nemuseli ukladať všetky objekty do databázy postupne, ale potrebovali sme, aby sa pri uložení napr. Requirement uložili do databázy aj jeho asociácie, teda Location a RequirementCompetence. Z tohto dôvodu sme anotáciám *@OneToMany*, *@ManyToOne* a *@ManyToMany* špecifikovali hodnotu atribútu *cascade* na CascadeType.**ALL**. *Objekty, ktoré využívajú vzťahy s inými objektami, sú častokrát závislé na objektoch z týchto vzťahov* [10]. Vďaka CascadeType môžeme určiť, ktoré zmeny sa vykonajú na týchto asociovaných objektoch. V tomto prípade som použil CascadeType.**ALL** z toho dôvodu, že pri všetkých vykonaných operáciách sa zmeny vykonajú aj na asociovaných objektoch tohto objektu. K možnosti ALL ešte existujú aj možnosti REMOVE, PERSIST, MERGE, DETACH a REFRESH. Vytváranie DAO bolo v tomto prípade rovnaké ako pri vytváraní DAO pre *data_adapter* časť. Teda bolo potrebné vytvoriť *repositories* a *services*, len s tým rozdielom, že rozhrania a implementácie services obsahovali aj metódy pre ukladanie a mazanie.

8.5 Kopírovanie dát

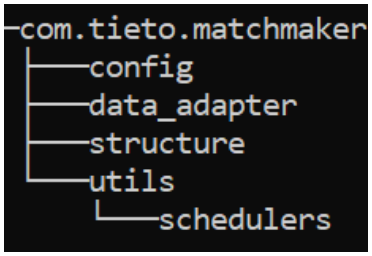
Zadanie

Potrebné zriadiť kopírovanie dát z centrálnej databázy spoločnosti do databázy aplikácie. Zariadiť aktuálnosť dát počas celého behu aplikácie.

Riešenie problému

Z dôvodu zachovania aktuálnosti dát, sme sa rozhodli pre také riešenie, vďaka ktorému sa budú dáta kopírovať z centrálne databázy každých 30 minút. Začal som tým, že som do štruktúry projektu pridal balíček *utils* (*pomôcky*) a v tomto balíčku som vytvoril balíček *schedulers* (*plánovače*), pozri obr. 8.

Do balíčku *schedulers* som pridal triedu DataCopyScheduler, ktorá bude slúžiť pre kopírovanie dát po štarte aplikácie a potom každých 30 minút. Ako prvé, bolo potrebné zariadiť aby táto trieda mala prístup k dátam z centrálnej databázy. Toto bolo zariadené jednoducho vytvorením asociácie na objekty tried RequirementDTOService a AdvertisementDTOService. Týmto bola



Obr. 8: Štruktúra projektu po pridaní utils balíčku

vytvorená možnosť kopírovania dát. Teraz ešte bolo potrebné spraviť možnosť pre ukladanie dát do databázy aplikácie. Podobne ako v predchádzajúcom prípade stačilo do triedy pridať objekty tried RequirementService a AdvertisementService. Taktiež bolo potrebné tieto objekty vytvoriť, to som spravil pomocou *dependency injection* vďaka anotácií *@Autowired* a tým, že som označil triedu anotáciou *@Component*. Anotácia *@Component* je genericky stereotyp pre akékoľvek Spring komponenty. Keď sa pozrieme bližšie na anotácie *@Repository* a *@Service*, zistíme, že sa jedná o špeciálne prípady použitia anotácie *@Component*, pozri výpis 5.

```
@Component
public @interface Service {}

@Component
public @interface Repository {}
```

Výpis 5: Anotácie @Service a @Repository

Keď bolo toto hotové, vytvoril som v triede DataCopyScheduler metódu *copyRequirements()*, ktorej úlohou bolo získať dáta z centrálnej databázy pomocou RequirementDTOService metódy *findAll()*, namapovať ich z objektov triedy RequirementDTO na Requirement a vložiť ich do databázy aplikácie. Mapovanie objektov z triedy RequirementDTO na Requirement je vykonávane pomocou jednoduchej triedy *SimpleMapper*, ktorá jednoducho podľa názvov premenných namapuje tieto objekty. Pred vložením do databázy aplikácie však museli byť predchádzajúce záznamy zmazané, preto som do rozhrania RequirementService pridal metódu *deleteAllThenSaveAll*(), ktorej úloha bola najprv všetky záznamy zmazať a potom uložiť nové. Aby nedošlo nejakou náhodou k problému, že sa dáta zmažú ale nové dáta sa nepodarí uložiť, tak som nad túto metódu v triede RequirementServiceImpl pridal anotáciu *@Transactional*, ktorá spôsobila to, že sa táto metóda správa ako transakcia, čiže ak by došlo k nejakému problému pri ukladaní alebo mazaní, dáta ostanú stále prístupné.

Po vytvorení metódy pre kopírovanie Requirements bolo ešte potrebné zriadiť to, aby sa táto metóda vykonala vždy po zapnutí aplikácie a potom podľa plánovača. Preto som vytvoril metódu *copyData()*, ktorej som pridal anotáciu *@Scheduled*, ako už hovorí názov anotácie, používa sa pre periodické vyvolávanie nejakej metódy. Anotácii *@Scheduled* som nastavil atribút *cron* na hodnotu **„0 0/30 * * * ?“**. *Cron* dovoľuje užívateľovi plánovať vykonávanie úloh periodicky

v špecifikovanom čase/dátume. *Cron* hodnota je udávaná nasledovne „sekundy minutá hodina deň-v-mesiaci mesiac deň-v-týždni rok <príkaz>", hodnoty rok a <príkaz> sú nepovinné. V prípade nášho cronu „0 0/30 * * * ?" to znamená, že sa úloha vykoná vždy v minútach 00 a 30, v sekunde 00, každú hodinu, každý deň v mesiaci, každý mesiac a „?" pre akýkoľvek deň v týždni. „*" znamená **každý**, „?" znamená **akýkoľvek**.

Toto však ešte k tomu, aby plánovač fungoval nestačilo. Posledné, čo bolo potrebné, bolo pridať anotáciu *@EnableScheduling* nad *Main class* (*hlavnú triedu*) aplikácie.

8.6 Koncové body aplikácie

Zadanie

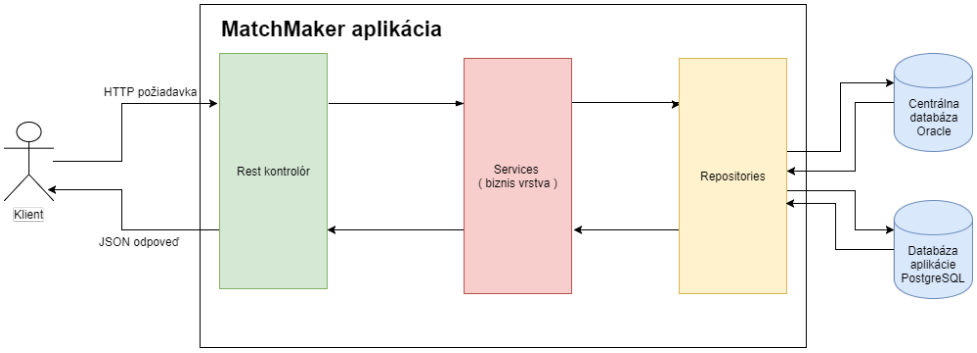
Aby bolo možné prijímať *http requests* (v preklade *požiadavky*) a odosielať *JSON response* (v preklade *odpovede*), je potrebné vytvoriť tzv. *kontrolóri*, ktoré definujú koncové body aplikácie pre prístup k dátam z "vonku", teda z front-end časti aplikácie.

Riešenie problému

K tomu, aby bolo možné získať na webe (front-end) dáta a pracovať s nimi, je potrebné tieto dáta z back-endu dostať pomocou koncových bodov.

Vytvoril som teda v projekte balíček *controllers*, v ktorom som vytvoril triedu s názvom RequirementController. Táto trieda bude slúžiť k práci s Requirement dátami. Pomocou kontrolóra dokážeme vyhotoviť metódy, ktoré dokážu obslúžiť všetky *http requesty – get, post, put, delete, options, head, patch, trace*. Requirement dáta chceme len získavať, čiže v tomto prípade nám stačí vytvoriť metódy pre *GET requests*. To sa pomocou Spring Framework robí veľmi jednoducho. Stačí vytvoriť metódu s príslušným návratovým dátovým typom, v našom prípade je to väčšinou *Pageable* a pridať tejto metóde anotáciu *@GetMapping* s hodnotou názvu koncového bodu, napríklad *@GetMapping("/find-all")* – k takejto metóde by bol prístup pomocou *<adresa-aplikácie>:<port-aplikácie>/find-all*. A však to by ešte nefungovalo, k tomu aby sme zaručili funkčnosť, treba tejto triede RequirementController pridať anotáciu *@RestController*, ktorá je pohodlnejším zápisom pre anotácie *@ResponseBody* a *@Controller*. Anotácia *@Controller* je špeciálny prípad použitia anotácie *@Component*, ktorá už bola niekoľkokrát spomenutá. Anotácia *@ResponseBody* spôsobí to, že objekt, ktorý je návratový z metód tejto triedy je automaticky serializovaný ako JSON a vložený späť do HttpResponse [11]. Potom som tejto triede pridal anotáciu *@RequestMapping* a tejto anotácii hodnotu "/requirements", vďaka čomu všetky koncové body tejto triedy majú ešte príponu "/requirements". Teda pre prípad spomenutý vyššie by to bolo *<adresa-aplikácie>:<port-aplikácie>/requirements/find-all*.

K tomu, aby bolo možné získať nejaké dáta z biznis vrstvy (*service layer*), je ešte potrebné pridať do tejto triedy RequirementService. Niečo podobné už sme robili ale niekoľkokrát, takže postup je veľmi jednoduchý, a to vďaka anotácii *@Autowired* a *dependency injection*.



Obr. 9: Vyhodnocovanie HTTP požiadaviek

8.7 Filtrovanie dát

Zadanie

Najdôležitejšou funkcionalitou aplikácie je filtrovanie. To znamená, že je potrebné vytvoriť možnosť rýchleho, efektívneho filtrovania s možnosťou kombinácie filtrov. Filtrovanie musí byť riešené spôsobom, že z prijatej http požiadavky dokáže zhotoviť potrebný filter, získať dáta podľa tohto filtra a odoslať ich pomocou koncového bodu v podobne JSON odpovede.

Riešenie problému

Z počiatku som začal implementáciu tohto filtru jednoduchým spôsobom a to tak, že som vytvoril metódu v rozhraní RequirementRepository, tejto metóde som pridal anotáciu *@Query*. Tejto anotácii môžeme nastaviť hodnotu SQL alebo JPQL, ktorá sa zavolaním tejto hodnoty vykoná v databáze. Z dôvodu, že takto sa síce dajú písať dotazy na databázu, ale nie je možné ich dynamicky upravovať a tým by sa filtrovanie podľa viacerých požiadaviek vykonávalo veľmi zložito a neefektívne.

Preto som sa rozhodol, aby mnou vytvorené rozhranie RequirementRepository rozširovalo rozhranie JpaSpecificationExecutor<T>, vďaka ktorému môžeme využívať Spring JPA Specification. Specification nám umožňuje využívať tzv. vyhľadávací jazyk na úrovni Java, čiže vďaka tomuto môžeme využívať Specification ako externý dómenovo špecifický jazyk (Domain-specific language).

Rozhodli sme sa, že požiadavky na filtrovanie budú v tvare, pozri tabuľka č. 1, a požiadavky budú oddelené čiarkou.

	kľúč	operácia	hodnota
Príklad	teamRole	==	Developer
Príklad	country	==	Poland

Tabuľka 1: Požiadavky na filtrovanie

Príklad toho ako bude vyzerať *query string* z http požiadavky zasielanej z front-end na koncový bod *search=“teamRole==“Developer“,country==“Poland““*. Tieto požiadavky sú pomocou Java triedy Pattern a pomocou regulárnych výrazov rozdelené na tri časti podľa požadovaného tvaru. Z tohto dôvodu som vytvoril triedu s názvom SearchCriteria, ktorá obsahovala presne tieto tri premenné - key(kľúč), operation(operácia), value(hodnota).

Ďalej som vytvoril triedu SpecificationBuilder<T>, ktorej úlohou je spojiť všetky požiadavky na filtrovanie, podľa návrhového vzoru Builder. Builder oddeľuje vytváranie objektov od ich reprezentácie, čo umožňuje ten istý proces vytvárania pre rôzne reprezentácie tohto objektu. Trieda je generická z toho dôvodu, že pre Requirements a Advertisements funguje filtrovanie rovnako. Vďaka tejto triede, môžeme kombinovať viac SearchCriteria dohromady. Použitie pre príklad spomenutý vyššie pozri výpis 6.

```
SpecificationBuilder<Requirement> builder = new SpecificationBuilder<>();
builder.with("teamRole", "==", "Developer")
      .with("country", "==", "Poland");
builder.build(Requirement.class);
```

Výpis 6: Príklad SpecificationBuilder<T>

Metóda *build(Class<T> forClass)* triedy SpecificationBuilder<T> prejde postupne všetky SearchCriteria, ktoré boli do tejto triedy pridané pomocou *with* metódy a vytvorí z nich List objektov abstraktnej triedy QuerySpecification<T> podľa parametra *forClass*, a potom všetky špecifikácie z tohto List-u spojí do jednoho objektu triedy Specification tak, že tento objekt obsahuje všetky vytvorené Specification. Toto celé je možné vďaka tomu, že abstraktná trieda QuerySpecification<T> rozširuje triedu Specification<T> a obsahuje konštruktor s parametrom SearchCriteria. Konečné filtrovanie je možné vďaka triede QuerySpecification<T>, kvôli tomu, že rozširuje Specification<T>, ktorá deklaruje metódu *toPredicate(. . .)* s návratovou hodnotou Predicate. Predicate je objekt, ktorý reprezentuje požiadavku na filtrovanie. Predicate objekty pre vyššie spomenutý príklad pozri výpis 7.

```
Predicate toPredicate(Root<T> root, CriteriaQuery query, CriteriaBuilder cb){
   return cb.equal(root.get("teamRole"), "Developer");
}

Predicate toPredicat(Root<T> root, CriteriaQuery query, CriteriaBuilder cb){
   return cb.equal(root.get("country"), "Poland");
}
```

Výpis 7: Príklad vytvárania objektu Predicate

Vzhľadom na to, že v aplikácii nebudeme filtrovať len podľa “equals”, ale aj podľa iných operácií, napr. väčšie, menšie a pod., tak som do metódy *toPredicate* pridal *if* bloky, ktoré podľa operácie zvolenej v objekte triedy SearchCriteria vytvoria správny Predicate.

To teda znamená, že metóda *build(Class<T> forClass)* triedy SpecificationBuilder<T> vráti objekt typu Specification, ktorý obsahuje všetky zvolené filtre. Tento objekt už stačí len jednoducho predať ako parameter do metódy RequirementRepository.*findAll(Specification<T> var1)* (rozšírenej z rozhrania JpaSpecificationExecutor<T>) a vďaka tomu získame dáta, ktoré odpovedajú zadaným filtrom.

9 Technológie využité pri implementácií Front-end

TypeScript

TypeScript je *open-source* programovací jazyk. V podstate sa nejedná o programovací jazyk ako taký, ale o nadstavbu nad programovacím jazykom JavaScript. TypeScript rozširuje JavaScript o statické typovanie a ďalšie atribúty objektovo orientovaného programovania. TypeScript kód sa po preložení kompiluje do JavaScriptu.

Pre TypeScript sme sa rozhodli z toho dôvodu, že vďaka nemu má programátor väčšiu kontrolu nad kódom a tým dokáže písať stabilnejší softvér.

ReactJS

ReactJS je v súčasnej dobe veľmi populárna knižnica k jazyku JavaScript (čiže aj pre TypeScript). Pomocou ReactJS je jednoduchšie písať znovu využiteľný kód, vzhľadom na to, že hlavnou výhodou ReactJS je rozdelenie kódu do znovu použiteľných modulov. Tým umožňuje programátorom využívať rovnaký modul viac krát.

Redux

Je knižnicou pre jazyk JavaScript. Redux je *open-source* knižnica pre riadenie stavu aplikácie. Veľmi dobre sa kombinuje s knižnicou ReactJS, preto sme sa rozhodli využiť kombináciu týchto dvoch knižníc.

Bootstrap

Bootstrap je ďalší z populárnych frameworkov pre vytváranie responzívnych, desktopových a mobilných web stránok. Bootstrap *framework* obsahuje balíčky CSS, slúžiacich pre úpravu stránok a rôznych komponent, ďalej obsahuje rozšírenie k jazyku JavaScript. My sme sa rozhodli využiť len balíčky obsahujúce CSS.

10 Riešené úlohy pri implementácii front-end

10.1 Získavanie dát z koncových bodov back-endu

10.1.1 Zadanie

Rozdeliť projekt do štruktúry tak, aby bolo možné v ňom využívať technológie TypeSript, ReactJS a Redux. Zriadiť jednoduché získavanie dát z koncových bodov.

10.1.2 Riešenie problému

Vzhľadom na to, že veľa komponent, ktoré budeme využívať, sú už vytvorené a naštýlované v projekte, ktorý využívame ako šablónu pre tvorbu front-end projektov, tak sme sa rozhodli práve túto šablónu využiť. V tejto šablóne sú len pridané základné komponenty, ako napr. tlačidlá, navigačné lišty a podobne.

Kvôli tomu, že budeme využívať Redux, sme sa rozhodli tak, že každá stránka bude vo vlastnom balíčku pomenovanom názvom tejto stránky. Tento balíček bude obsahovať 5 TypeScript súborov – index.ts, constants.ts, actions.ts, sagas.ts, reducer.ts. V súbore index.ts je obsiahnutý spôsob vykresľovania elementov do stránky, sú tu zvolené využité elementy, je možné tu voliť štýly pre dané elementy a určovať spôsob vykresľovania, tento súbor je v podstate to, čo užívateľ uvidí. Zvyšné štyri súbory, teda constants.ts, actions.ts, sagas.ts, reducer.ts sú súčasťou Redux, a teda ich úlohou je riadenie stavu aplikácie.

Začal som tým, že som najprv vytvoril balíček RequirementsContainer a v tomto balíčku som vytvoril prvý súbor constants.ts. Tento súbor bude slúžiť ako zoznám zadefinovaných konštánt, podľa ktorých bude možné ďalej v aplikácii volať správne akcie a správne mapovať dáta na stav aplikácie. V tomto súbore som vytvoril zatiaľ len jednu konštantu s názvom FETCH_REQUIREMENTS, ktorá bude slúžiť práve pre vyvolávanie akcií a mapovanie pre získanie *requirements* dát. Hodnota tejto konštanty je ľubovoľná, ale musí byť jedinečná. Ďalej som v tomto súbore vytvoril konštantu cReducer, ktorej som pridal hodnoty, ktoré budú reprezentovať kľúče k dátam v *state* aplikácie. V tomto prípade to teda bolo *requirementsData*, ktoré bude reprezentovať kľúč k *requirements* dátam a hodnota *loading*, ktorá bude reprezentovať či sú dané dáta načítané alebo nie.

Ďalej som vytvoril súbor actions.ts. V tomto súbore som vytvoril konštantu s názvom *fetchRequirements*. Táto metóda vyvoláva funkciu *action* z balíčku Redux. Tejto funkcii *action* sú predané parametre FETCH_REQUIREMENTS zo súboru constants.ts a prázdny objekt. Toto v jednoduchosti znamená to, že sa vyvolá v stave aplikácie konštanta FETCH_REQUIREMENTS a na ňu zareaguje sagas.ts a reducer.ts. Druhý parameter funkcie *action* je používaný pre zasielanie dát do sagas.ts, to ale v tomto prípade nepoužívame, pretože to pre nás nemá žiaden význam.

Potom som vytvoril súbor sagas.ts. Ságy sú v podstate funkcie, ktoré by mali ovládať všetku biznis logiku front-end časti aplikácie. V tomto prípade som vytvoril ságu s názvom *fetchRequirements*. Pri vytváraní ságy sa používa nasledovný zápis, pozri výpis č. 8.

```
export function* fetchRequirements(action: any){ ... }
```

Výpis 8: Ukážka vytvárania sága funkcie

Hviezdička(*) za kľúčovým slovom *function* sa využíva pre to, aby sme mohli v metóde využiť *yield*. V tejto metóde *fetchRequirements* som vytvoril konštantu *response*, ktorej hodnota budú dáta z back-endu. Tieto dáta získame tým, že zavoláme koncový bod back-end aplikácie. V tomto prípade nám ide o koncový bod "/requirements". Spravil som teda to, že som pridal hodnote response volanie na koncový bod „/requirements" s GET požiadavkou. Ďalej som spravil to, že túto hodnotu *response* som predal ako parameter funkcií *put* z balíčku Redux, pozri výpis č. 9.

```
yield put(fulfilled(FETCH_REQUIREMENTS, response));
```

Výpis 9: Sága volanie put metódy

Týmto spôsobím to, že na toto volanie zareaguje reducer.ts a pomocou zavolanej konštanty FETCH_REQUIREMENTS budem môcť v tomto reduceri nastaviť čo sa má s dátami vykonať. Ďalej som v tomto súbore sagas.ts musel nastaviť to, aby tento súbor vedel, ktorú ságu ma vyvolať pri konštante FETCH_REQUIREMENTS. To som spravil tak, že som vytvoril funkciu, ktorá v podstate funguje ako mapa ság, s tým, že kľuč je názov konštanty a hodnota je sága, ktorá sa má vyvolať.

Vytvoril som reducer.ts. Reducer je tá časť aplikácie, ktorá zodpovedá za udržiavanie a riadenie jej stavu. V tomto reduceri som vytvoril *switch*, ktorý reaguje na zasielane konštanty. Vytvoril som tu teda *case* pre FETCH_REQUIREMENTS a FETCH_REQUIREMENTS FULFILLED. Možno je trochu divné, prečo som vytvoril práve dva, keď máme len jednu konštantu FETCH_REQUIREMENTS. Ale každý z týchto *case-ov* slúži na niečo iné. Prvý prípad, teda FETCH_REQUIREMENTS, sa zavolá ihneď po zavolaní metódy action v súbore actions.ts. To znamená, že v momente tohto *case* sa ešte nevyvolala žiadna sága. V tomto *case* nastavím hodnotu cReducer.*loading* na *true*, pre indikovanie toho, že dáta sa načítavajú. Druhý, teda FETCH_REQUIREMENTS FULFILLED sa vykoná po ukončení metódy (ságy) zo súboru sagas.ts a v tomto momente už budú prístupne dáta z back-endu. V tom prípade ich teda namapujem na vytvorený objekt v našom projekte a uložím ich do stavu aplikácie ako cReducer.*requirementsData*.

Ako posledné som vytvoril index.ts, v ktorom som vytvoril triedu RequirementContainer, ktorá rozširuje triedu React.Component s typom RequirementContainerProps. Použité rozhranie RequirementContainerProps som vytvoril v tomto súbore. Toto rozhranie obsahuje parametre, ktoré sú potrebné alebo voliteľné pri vytváraní tejto triedy. Do tohto rozhrania som pridal RequirementContainerReducer, ktorý reprezentuje *reducer* z reducer.ts. Potom som doň pridal *actions*, ktoré bude reprezentovať akcie zo súboru actions.ts. Nakoniec som pridal namapovanie

týchto actions a reduceru na tento kontajnér pomocou metódy *connect* z balíčku Redux. Keď bolo toto hotové, už bolo možné písať priamo kód pre vykresľovanie stránky.

Ako prvé som vytvoril na tejto stránke tlačidlo. Pre vytvorenie tlačidla na stránke, je nutné v React.Component rozširujúcej triede pridať metódu render a tejto metóde špecifikovať, čo má vlastne vykresliť. Ďalej som pridal tomuto tlačidlu možnosť *onClick*, to znamená, že vždy po kliknutí na toto tlačidlo sa vyvolá metóda, ktorá je špecifikovaná. Vytvoril som metódu *fetchRequirement* a pridal som jej volanie tomuto tlačidlu. V tejto metóde som volal metódu *fetchRequirements* zo súboru actions.ts, pre to aby som otestoval, či všetko funguje tak ako má, pozri výpis č. 10.

```
class RequirementContainer extends React.Component<RequirementContainerProps>{
private fetchRequirements() {
    this.props.actions.fetchRequirements();
}
public render(){
   return (
      <button onClick={() => this.fetchRequiremenets()}>BUTTON</button>
   );
}
}
```

Výpis 10: Trieda RequirementContainer

10.2 Základne zobrazenie dát

Zadanie

Vytvoriť jednoduché UI pre zobrazenie dát, tak aby bolo možné dobre vidieť všetky potrebné dáta a neskôr v nich jednoducho vyhľadávať podľa kľúčových vlastností.

Riešenie problému

Vzhľadom na to, že bolo potrebné zobraziť veľké množstvo dát na jednej stránke, tak aby boli dobre čitateľné a dobre sa v nich vyhľadávalo, sme sa rozhodli, že dáta budú reprezentované v tabuľkovej forme. Preto som v súbore index.ts v RequirementsContainer vytvoril tabuľku. Jedná sa o jednoduchú HTML tabuľku, s tým že táto tabuľka neobsahuje pri vytváraní žiadne dáta, iba hlavičku tabuľky.

Aby som mohol ďalej pokračovať vo vysvetľovaní, musím najprv vysvetliť životný cyklus(*lifecycle*) react komponentu. Vytváranie react komponentu prebieha následovne. Po štarte vytvárania komponentu najprv prebehne konštruktor komponentu – *constructor(pros)*. Ďalej nasleduje *componentWillMount* metóda a po nej nasleduje *render* metóda. Po *render* metóde nasleduje metóda *componentDidMount* a vtedy už je komponent plne vykreslený. Ďalej react

komponent obsahuje cyklus *updating*, teda pri zmene stavu komponentu alebo aplikácie. Tento cyklus je robený tak, aby komponent zareagoval na zmeny stavu a prekreslila sa.

Na to, aby som dostal do tabuľky nejaké dáta, som ich musel najprv získať. Preto som vytvoril metódu *componentWillMount* a v tele tejto metódy som zavolal akciu *fetchRequirementsData*, ktorú som už spomínal. Tým, že je táto akcia volaná v metóde *componentWillMount* docielime to, že ešte pred vykreslením komponentu sa zašle žiadosť o dáta na back-endový koncový bod a počas vykreslovania už bude komponenta obsahovať potrebné dáta. Ďalej som spravil to, že som do *render* metódy pridal časť kódu na získavanie dát z reducer.ts pomocou *this.props.requirementContainerReducer*, ktorý je uvedený v RequirementContainerProps, a teda pri vytvorení komponentu je tento reducer v spojení s týmto komponentom. Získať dáta zo stavu tohto reducera je jednoduché a robí sa to pomocou konštanty *cReducer*, ktorú som definoval v constants.ts. Pre získavanie dát z reduceru pozri výpis č.11.

```
const data = this.props.requirementContainerReducer.get(
   cReducer.requirementsData
);
```

Výpis 11: Získavanie reducer dát

Keď už máme získané tieto dáta, tak pomocou metódy *map*, ktorú je možné v JavaScripte aplikovať na pole(*array*), namapujeme tieto dáta na riadky v tabuľke a uložíme ich do dočasnej konštanty, ktorú potom v metóde render vykreslíme, pozri výpis č.12.

```
const rows = data.map((requirement) -> {
   return (<tr>
       <td>requirement.teamRole</td>
       <td>requirement.startDate</td>
       <td>...</td>
   </tr>)
});
```

Výpis 12: Vytváranie riadkov tabuľky z dát

10.3 Filtrovanie podľa základných kľúčov

Zadanie

Vytvoriť jednoduché filtrovanie tabuľky, vďaka ktorému sa uživateľovi budú zobrazovať dáta, podľa zadaného filtra. Tento filter bude reprezentovaný v hlavičke tabuľky.

Riešenie problému

Jednoducho som pretvoril hlavičku tabuľky tak, aby sa po kliknutí zmenila na vstupné pole pre vstup od uživateľa. Štýlovanie a podobné veci tu popisovať nebudem vzhľadom na to, že

nie sú tak zaujímavé. Keď som mal vytvorené hlavičky tabuľky, ktoré mohli slúžiť ako políčka pre vyhľadávanie, tak bolo potrebné vytvoriť k vyhľadávaniu *constants, actions, sagas a case v reducer*. Toto všetko som spravil veľmi podobne ako pre prípad *fetchRequirements*, ktorý som už spomínal, len s tým rozdielom, že som teraz do actions *fetchRequirementsBySearch* pridával parameter ktorý obsahoval dáta, ktoré sú vlastne mapa vyhľadávacích políčok z hlavičky tabuľky, teda ich názov a text zadaný do políčka od užívateľa. Tieto dáta boli vo formáte podobnom tomuto: teamRole:"Java deve". Tu už môžeme vidieť podobnosť s tým, aké požiadavky treba zasielať na koncové body back-endu. V *action* som s týmto parametrom nič nerobil, iba som ho zasielal ďalej do *sagas*. V ságe som tieto dáta upravil do formátu tak, aby sa zhodoval s back-endom. Teda ak som chcel, aby výsledne vyfiltrované dáta obsahovali *"team role"* s hodnotou *"Java deve"*, tak sa v ságe tento vyhľadávací reťazec upravil na teamRole=="Java deve". Keď som mal hotové to, aby sa vytvoril tento vyhľadávací reťazec, tak ho stačilo pridať ako parameter ku koncovému bodu back-endu. Tento koncový bod vyzeral teda ako *„/requirements/filter?search="teamRole=="Java deve","”*. Proces namapovania dát na stav aplikácie v reducer.ts prebiehal rovnako ako v prípade *fetchRequirements*.

Team role	Requirement state	Avg.time	Country	City	Proficiency	Start date	End date	Published	Location mandatory
Web Developer	04 Searching For Candidates	8	Czech Republic		EXPERIENCED	01.04.2019	31.12.2019	15.03.2019	
PLaV_8: Web Developer	04 Searching For Candidates	8	Finland	Helsinki Metropolitan...	EXPERIENCED	01.05.2019	31.01.2020	15.03.2019	✓
ND Business Project Managers	04 Searching For Candidates	8	Norway	Oslo	EXPERIENCED	25.03.2019	28.06.2019	15.03.2019	
Test Engineer	05 Evaluation Of Candidates	8	India	Pune	EXPERIENCED	15.03.2019	31.12.2019	15.03.2019	✓

Obr. 10: Zobrazené dáta bez filtru

Java deve	Requirement state	Avg.time	Country	City	Proficiency	Start date	End date	Published	Location mandatory
Java Developer	04 Searching For Candidates	8	India	Pune	SENIOR	01.02.2019	31.12.2019	14.03.2019	✓
Java Developer	04 Searching For Candidates	8	India	Pune	SENIOR	01.02.2019	31.12.2019	14.03.2019	✓
Java Developer	05 Evaluation Of Candidates	8	India	Pune	EXPERIENCED	22.02.2019	29.02.2020	22.02.2019	
PL/SQL With Java Developer (R...	07 Recruitment	8	India	Pune	EXPERIENCED	18.02.2019	31.12.2019	18.02.2019	✓

Obr. 11: Zobrazené dáta s filtrom

11 Hodnotenie znalostí potrebných počas praxe

11.1 Využité znalosti zo štúdia

Vzhľadom na to, že celý back-end bol postavený na programovacom jazyku Java, tak som využil znalosti z predmetu Programovací jazyky I pre základnú konštrukciu jazyka Java. Potom som využil znalosti z predmetu Java Technológie pre využitie JPA. Ďalej som využil znalosti z databázových predmetov, vzhľadom na to, že celá aplikácia je postavená na práci s databázou a vyhľadávaním v tejto databáze. Ďalej som využil znalosti z predmetu Tvorba aplikácí pro mobilní zařízení I pre prácu s JavaScriptom a základmi HTML a CSS. Znalosti z predmetu Tvorba aplikací pro mobilní zařízení II pre základy s verzovacím systémom GIT. Ďalej som využil znalosti z predmetu Úvod do softwarového inženýrstvi, z ktorého som využil znalosti vytvárania diagramov pri návrhu systému a tiež pri návrhu architektúry aplikácie. Ďalej som využil znalosti z predmetu Vývoj informačních systému pri návrhu implementácie a architektúry aplikácie.

11.2 Chýbajúce znalosti zo štúdia

Určite by som uvítal vo viacerých predmetoch prácu s verzovacím systémom GIT, vzhľadom na to, že práca s týmto systémom je veľmi dôležitá pri práci na projektoch s väčším počtom ľudí. Taktiež by som uvítal projekty kde musí pracovať na projekte viac ľudí, pre osvojenie znalosti pracovania v tíme. Ďalej mi počas praxe chýbali znalosti práce s http requestami, popis rest architektúry. Taktiež som sa na praxi prvýkrát dopočul o Spring Framework, o ktorom som sa musel veľmi veľa počas praxe doučiť. Taktiež mi chýbali znalosti o technológii Maven, ktorý je veľmi využívané pre uľahčenie budovania a nasadzovania aplikácií a celkového riadenia aplikácie. O tejto technológii som sa prvýkrát dozvedel až počas bakalárskej praxe.

12 Záver

Prax v spoločnosti Tieto Czech s.r.o. hodnotím veľmi pozitívne, pretože mi priniesla veľmi veľa skúseností potrebných pri vývoji profesionálnych projektov a aplikácií v praxi. Naučil som sa o nových frameworkoch a pracovných postupoch, ktoré sú zaužívané v IT spoločnostiach. Taktiež som sa naučil používať verzovací systém GIT a pracovať v tíme viacerých programátorov. Mal som možnosť pracovať po boku skúsených programátorov, ktorí mi pomohli zlepšiť moje znalosti v oblasti písania kvalitného kódu a celkovo návrhu systému. Vďaka praxi som pochopil ako funguje vývoj projektov vo veľkej nadnárodnej korporácií. Taktiež som zistil ako v praxi funguje agilná metodika SCRUM. Tiež som mal možnosť pracovať a podieľať sa na vývoji zaujímavých projektov.

Vďaka absolvovanej praxi som nazbieral mnoho skúsenosti, ktoré verím, že dokážem uplatniť naďalej v mojom kariérnom ale aj osobnom živote.

Literatúra

[1] O Tieto [online]
Informácie dostupné na: `https://www.tieto.com/cz/about-us/our-company/`
Citované 8.3.2019

[2] Historia Tieto [online]
Informácie dostupné na: `https://campaigns.tieto.com/tieto50/#`
Citované 13.3.2019

[3] Jazyk Java [online]
Informácie dostupné na: `https://www.oracle.com/technetwork/java/index.html`
Citované 16.3.2019

[4] Databáza [online]
Informácie dostupné na: `https://www.websupport.sk/faq/co-je-to-databaza`
Citované 16.3.2019

[5] Sprign Framework [online]
Informácie dostupné na: `https://spring.io/projects/spring-framework`
Citované 16.3.2019

[6] Apache Maven [online]
Informácie dostupné na: `http://maven.apache.org/what-is-maven.html`
Citované 16.3.2019

[7] HibernateORM [online]
Informácie dostupné na: `http://hibernate.org/orm/what-is-an-orm/`
Citované 16.3.2019

[8] Spring Dokumentácia - Repository [online]
Informácie dostupné na: `https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Repository.html`
Citované 10.3.2019

[9] Spring Dokumentácia - ApplicationContext [online]
Informácie dostupné na: `https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/ApplicationContext.html`
Citované 10.3.2019

[10] Oracle Dokumentácia - Kaskádove operácie a vzťahy [online]
Informácie dostupné na: `https://docs.oracle.com/cd/E19798-01/821-1841/bnbqm/index.html`
Citované 11.3.2019

[11] Baeldung - Spring Response body [online]
Informácie dostupné na: `https://www.baeldung.com/spring-request-response-body`
Citované 12.3.2019